Introduction to the Interactor® language.

Mark Coniglio

The Center for Art, Information and Technology
California Institute of the Arts
24700 McBean Parkway
Valencia, California 91355

ABSTRACT

Interactor, a graphically based programming language developed for the Macintosh by Mark Coniglio and Morton Subotnick over a four year period, is devoted to facilitating real-time interactions between live performers and computer. The most important objective was to develop an overall structure which addressed the linear but highly flexible nature of pre-composed performances. Only slightly less important were an effective user interface and absolute minimum response times. This paper gives a brief description of the language and strategies for its use in live performance.

1. HISTORY

Interactor was begun in response to experiments Mort Subotnick has done concerning tempo following during a residency at the Massachusetts Institute of Technology in 1985. Using David Levitt's HookUp! program to model various parameters of a performance, Subotnick was attempting to solve problems of score following, that is, the ability of the computer to locate musical gestures based on input by a live performer within a pre-composed score. One area in which he had some measure of success was in measuring the tempo of notes arriving from a MIDI keyboard. This led to the following hypothesis: being able to accurately measure the tempo at which a performer is playing is the primary information needed to follow a performer through the score.

This hypothesis exhibited the following benefits. First was the low computing overhead required to measure the performer's tempi – much less than the power of the simplest pattern matching algorithms, which attempt to match the pitches played by a performer with pitches in a score. Second, the filtering of erroneous data is extremely simple. By specifying ranges of tempi allowed for a given section, tempi outside of this range could be ignored. "Fluffed" notes are simply discarded. Further, it was easy to imagine software that would allow the performer to "rehearse" with the computer so that it would learn the tempo at each moment of the piece, thus improving reliability and allowing for sudden changes of tempo.

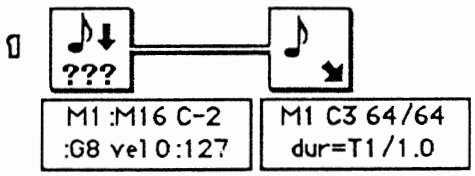
Upon his return to the California Institute of the Arts, the author, who was then a student, was engaged by Subotnick to create a piece of software for the Macintosh that would allow the most basic investigation of these concepts. This would, over a four-year period, evolve into the Interactor language.

2. LANGUAGE OVERVIEW

2.1 Structure and Syntax

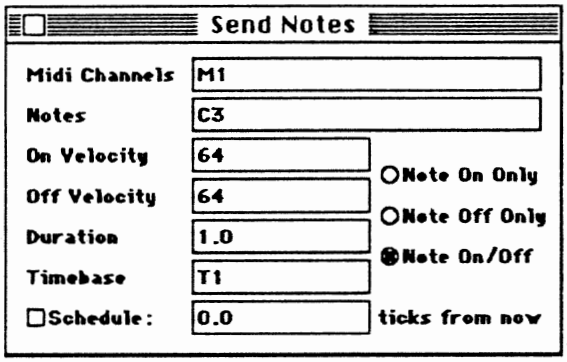
It is clear that real time interaction must be driven by events. We wanted to create a language that was predisposed to responding to these events within the context of a pre-composed score. This being true, the decision was made to impose an organizational structure which would allow language statements to be conveniently grouped into what are called *Scenes*. Each Interactor program can have several scenes, each of which dictates its own logic. An ordered list of scenes is known as a *Scene Group*. As with a play (at least traditional ones) only one scene is active at any given moment in time. When an event occurs, it is sent to the active scene for processing.

Each statement in a scene is built of the smallest language unit, *operators*, which are represented by graphic icons. Operators fall into two broad categories: those that test conditions and those that perform functions. Operators that test a condition will pass control to the operator immediately to the right if the condition is true, allowing execution to continue. Otherwise, control passes to the statement below, if one exists. Operators that perform functions always pass control to the operator to the right. Below is a an illustration of a two operator statement.



The first operator tests for a Note On message. When the Note On event occurs, control will pass to the second operator which immediately sends a MIDI Note On message with a pitch of C3, and schedules a corresponding Note Off to be sent one quarter note into the future.

All of the information required to execute an operator is embodied within that operator. By double-clicking on an operator's icon, a window will open which allows the user to edit that operator's parameters. Here is an example of the parameter window for the operator which sends MIDI notes.



The structure of operator within a statement may appear simplistic, but it has the two major benefits. First, no data is passed from operator to operator. This means that no matter how complex a Scene is, the processing overhead due to passing messages and data remains nil, insuring extremely fast responses to incoming events. Second, the left to right, top to bottom structure of statements provides a very clear visual representation of the logical structure of the scene.

2.2 Flexible Time

It was theorized above that the accurate measurement of time was an extremely reliable and economical method of determining a performer's location within a score. To this end, Interactor supports eight independent sources of timing (Timebases) which are specified in beats per minute. Any operator which is needs timing information can reference any of the eight timebases. This is an important departure from other interactive software programs which choose to base measurements of time on an inflexible unit (i.e., milliseconds.) Also, several operators devoted to the measurement of tempo are provided. The usefulness of flexible time is shown in the following example.

Consider this situation: The score calls for the computer to accompany the live performer starting on the downbeat of measure 4. In 4/4 time, this is 12 quarter notes from the beginning of the piece. At a tempo of 90 BPM we know that the downbeat of measure 4 will occur 8 seconds (12 beats * .66666 seconds per beat) from the start of the piece. We could schedule the computer to join in exactly 8

seconds from the first note played. It is highly likely that the performer will stray (at least slightly) from the dictated tempo of 90 BPM, causing the computer to be out of sync with the performer at measure 4. On the other hand, if we follow the score using a method of tempo following, the performer can choose *any* tempo, can make several errors in the pitches played, and the computer will still begin playing at the correct moment.

This bias towards time than can expand or shrink to match the live performer is one of Interactor's most useful design features. In addition to the score following possibilities mentioned above, it means that you can schedule an event to occur or a series of operators to execute at a time *n* beats into the future. As long as accurate tempo following is maintained, the scheduled item will occur in perfect synchronization with the performer.

2.3 Other Features

One operator is specifically designed to let you pass parameters to another scene and execute it as a subroutine. This serves to reduce the duplication of code and to organize the programs more efficiently.

Interactor supports the playback of up to 256 multi-channel sequence tracks, which may be imported from standard MIDI files. Initiation and cessation of playback may be independently controlled for each sequence. Tempo may be derived from any of the eight Timebases. Operators provide non-destructive transposition and velocity modulation of Note On and Off messages, location, looping and many other sequencer based features.

Lists of numbers may be stored and accessed during program execution, or may be created and maintained out of real time. Operators allow insertion and deletion of items, the modification of previously stored values, searching capabilities, etc. Both graphic and item based editing (e.g., a list of numeric values or note names) is offered.

The programmer may create the user interface using several pre-defined Controls. These include faders (both rotary and linear,) numeric values, pop up menus, various switches, sequence track monitors, MIDI activity monitors, etc.

2.4 Development information

Original versions of Interactor were written in Pascal. During the last major revision, a switch was made to the Lightspeed C environment. This was done for the following reasons: 1) C induces less language related overhead than Pascal, 2) the availability of inline assembly language in C, and 3) the Think Class Library provided as part of the Think C package. This object-oriented implementation of the most needed functions required by the Macintosh user interface simplified development tremendously.

3. A STRATEGY FOR USE IN LIVE PERFORMANCE

Interactor can be useful in both an improvisational and pre-composed context as a real-time remapper, modulator, or extender of events. Since it was first intended to be used with a live performer playing written music, the example that follows documents the implementation of a score follower. Here the performer plays the right hand of a solo piano piece (specifically a Debussey prelude) while the computer accompanies with the left hand. In addition, the program allows the performer to start at almost any measure in the piece.

The most important step is the analysis of the score. Here, the composer/programmer looks for unique combinations of pitches and or rhythms to be used as "landmarks" or anchoring points when determining the location of the performer within the score. For this example, I looked for unique

dyads and the rhythmic distance between them. This information is specific enough to specify many unique combinations for this type and length of piece.

Once the dyads have been chosen, it is a simple matter to program the piece. One statement reads the incoming dyads and measures rhythmic distance between them. It also uses the Follow Tempo operator to measure the tempo and then adjust the tempo one of the Timebases. Finally, it uses the velocity of the incoming Note On messages to modulate the velocity of the Note Ons in the left-hand accompaniment played by the sequence track.

The remaining statements are all very similar: compare the most recent dyad information with that of one of the analyzed dyads. If there is a match, jump to that point in the score and begin playback of the left hand sequence.

This example may seem trivial, but it nevertheless works extremely well. Once the program has located its position in the score, the performer can make many mistakes – the computer will simply follow his or her tempo and continue playing. This can happen precisely because of the low reliance on pitch material by this score following algorithm. This resiliency to errors is critical when actually performing in a concert hall for an audience.

One interesting aspect of this algorithm is that the analysis portion could very probably be automated, though human abilities at recognizing patterns during the analysis phase are not to be underestimated.

4. FUTURE ENHANCEMENTS

Of the several improvements to be implemented, foremost is the addition of my own MIDI drivers. The original version allowed only for use with Apple's MidiManager™. This was a mistake for two reasons: the high overhead of the MidiManager adversely affected performance on lower grade Macintoshes (Plus, SE, Classic) and my unfounded belief that Apple would actually be committed enough to the MidiManager to spend time improving it as time progressed. These drivers are being beta-tested at the time of this writing. On a related note, some users have requested OMS compatibility. This will be taken under consideration in the coming months.

The ability to lock the timebases to MIDI time code or to a MidiManager time port has been requested by many users. This would allow Interactor to work in synchronization with other MidiManager compatible programs. This will feature will be added by the time this paper is presented.

The next major revision will include the ability to create libraries of useful scenes which can be accessed as a subroutine. It will also include the ability to edit the operators as text, allowing the implementation of a search and replace feature.

5. ACKNOWLEDGEMENTS

Interactor could not have been written without the advice and insight of Morton Subotnick, to whom I am very grateful.